National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

TECHNICAL MEMORANDUM 97/231
June 1996

TRANSOM: A MULTI-METHOD REYNOLDS-AVERAGED NAVIER-STOKES SOLVER: OVERALL DESIGN

David Hally

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

9 GROVE STREET
P.O. BOX 1012
DARTMOUTH, N.S.
B2Y 3Z7

TELEPHONE
(902) 426-3100

CENTRE DE RECHERCHES POUR LA DÉFENSE ATLANTIQUE

9 GROVE STREET
C.P. BOX 1012
DARTMOUTH, N.É.
B2Y 3Z7

National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

TRANSOM: A MULTI-METHOD REYNOLDS-AVERAGED NAVIER-STOKES SOLVER: OVERALL DESIGN

David Hally

June 1996

Approved by R.W. Graham:
Head/Hydronautics Section

TECHNICAL MEMORANDUM 97/231

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

TRANSOM is a multi-block, multi-method Reynolds-Averaged Navier Stokes solver being developed at DREA to address problems associated with the flow around ships and submarines. It is multi-block because the flow is divided into several distinct regions. It is multi-method because a different solution method may be used on each of the flow regions. At present two different methods of solution can be chosen: a finite-volume solver based on the pseudo-compressibility method; and a finite element solver which uses the penalty function method to determine the pressure.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the overall design of TRANSOM with emphasis on the class hierarchies used to represent different types of blocks and flow solvers. The algorithms used to implement the pseudo-compressibility and finite element methods are not discussed here; two companion reports describe these sub-solvers in detail.

Résumé

TRANSOM est un résolveur Navier-Stokes pondéré Reynolds, multiméthode, multibloc, que le CRDA est en train de développer pour résoudre des problèmes concernant l'écoulement autour des navires et des sous-marins. Il est multibloc parce que l'écoulement se divise en plusieurs régions distinctes. Il est multiméthode parce qu'il permet d'utiliser une technique de solution différente pour chacun des régions de l'écoulement. A l'heure actuelle on peut choisir entre deux méthodes de solution: un résolveur à volumes finis basé sur la méthode de pseudo-compressibilité et un résolveur par éléments finis qui utilise la méthode de la fonction de pénalité pour déterminer la pression.

TRANSOM a été écrit en C++ en suivant les principes de la programmation orientée objet. Ce document décrit la conception générale de TRANSOM et met l'accent sur les hiérarchies de classe utilisées pour représenter différents types de blocs et de résolveurs d'écoulement. Les algorithmes utilisés pour mettre en oeuvre les méthodes de pseudo-compressibilité et par éléments finis ne sont pas traités ici. Deux rapports d'accompagnement décrivent ces sous-résolveurs en détail.

DREA TEchnical Memorandum TM/97/231

TRANSOM

A Multi-Method Reynolds-Averaged Navier-Stokes Solver

Overall Design

by

David Hally

Executive Summary

Background

The numerical prediction of marine propeller performance and cavitation requires an accurate prediction of the nominal wake: that is, the flow into the propeller plane without the influence of the propeller itself. At DREA, the nominal wake is currently calculated with the HLLFLO programs which use a panel method to predict the potential flow and a boundary layer method to predict the viscous flow.

The HLLFLO programs have the advantage that they are very fast. They yield fairly good predictions for unappended destroyer hulls; however, there are several areas in which the predictions can be improved.

1. As is typical of boundary layer methods, the wake deficit is over-predicted near the stern. This is especially noticeable at model-scale Reynolds numbers.
2. HLLFLO is incapable of handling the flow past appendages. The wakes shed from the propeller shafts and shaft brackets can cause significant spatially concentrated wake defects which, in turn, affect higher harmonics of the radiated propeller noise and the speed at which cavitation first occurs.
3. HLLFLO does not account for the influence of the boundary layer on the potential flow.

These effects are especially important for smaller vessels with low length/beam ratios. Moreover, HLLFLO is incapable of handling the complex geometries of submarines or advanced marine vehicles such as small water area twin hull (SWATH) ships.

Principal Results

The program TRANSOM has been developed address these deficiencies. TRANSOM is a multi-block, multi-method Reynolds-Averaged Navier Stokes solver; multi-block because the flow is divided into several distinct regions called blocks; multi-method because a different solution method may be used on each of the blocks.

The current version of TRANSOM includes two different solution

methods which can be used on the blocks. The first is a pseudo-compressibility method based on the work of Rogers and Kwak; it is suitable for use on structured blocks. The second uses the finite element method and is based on the program MEF developed at Université Laval; it is suitable for use on unstructured blocks. Turbulence may be modelled using variants of the k-ε model or by the Baldwin-Lomax model. Currently only two-dimensional flow can be modelled.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the overall design of TRANSOM and, in particular, the class hierarchies used to represent different types of blocks and flow solvers. The algorithms used to implement the pseudo-compressibility and finite element methods are not discussed here; two companion reports describe these sub-solvers in detail.

Significance of Results

When TRANSOM is fully developed it will provide accurate numerical predictions of the flow into the propeller plane. It will also be applicable to many other flow problems related to ships: for example, flow around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, prediction of roll damping coefficients, and prediction of the lift and drag characteristics of lifting surfaces such as submarine control planes, rudders, and fin stabilizers.

Future Work

TRANSOM is still under development; the current version will only solve two-dimensional flows and requires upgrading in many areas, in particular the ways in which different solution methods interact at their common block boundaries. Further development will be done at DREA and under contract.

Table of Contents

List of Figures

1 Introduction

At high speeds the noise radiated by a warship is primarily due to cavitation on its propellers. Naval propellers are designed so that cavitation inception is delayed to as high a speed as possible, but the design method requires an accurate prediction of the flow into the propeller plane. Over the past decade computer programs have been developed at DREA to predict the flow into the propeller plane. Known collectively as HLLFLO[1], these programs use a low-order panel method to predict the potential flow and a boundary layer method to calculate the viscous flow.

The HLLFLO programs have the advantage that they are very fast. They yield fairly good predictions for unappended destroyer hulls[2]; however, there are several areas in which the predictions can be improved.

1. As is typical of boundary layer methods, the wake deficit is over-predicted near the stern. This is especially noticeable at model-scale Reynolds numbers.
2. HLLFLO is incapable of handling the flow past appendages. The wakes shed from the propeller shafts and shaft brackets can cause significant spatially concentrated wake defects which, in turn, affect higher harmonics of the radiated propeller noiseand the speed at which cavitation first occurs.
3. HLLFLO does not account for the influence of the boundary layer on the potential flow.

These effects will be exacerbated by the current trend toward smaller vessels with smaller length/beam ratios. Moreover, HLLFLO is incapable of handling the complex geometries of submarines or advanced marine vehicles such as SWATHs.

The program TRANSOM is intended to address these deficiencies. TRANSOM is a Reynolds-Averaged Navier-Stokes (RANS) solver: in conjunction with a turbulence model, it solves the Navier-Stokes equations on a grid of points which cover the regions of flow.

The chief practical problem in developing a Navier-Stokes solver for the flow past a ship is that the Reynolds Number is so high: for a destroyer moving at 20 knots the Reynolds number is about 10^9. The higher the Reynolds number, the more grid points are necessary to resolve the severe velocity gradients near solid surfaces. For this reason, issues of computer storage are of extreme importance.

Because of the complex geometries of appended surface ships or submarines, the type of grid used in the calculations is also important because it affects the amount of computer storage required. Methods which allow the use of unstructured grids (e.g. finite element solvers) typically require far greater computer storage than methods which require structured grids. Thus, there is a trade-off between the ease generating the grid, and the storage requirements. To address this problem, TRANSOM has been designed as a

multi-block multi-method solver. The numerical grid may consist of several blocks, some of which are structured, some unstructured. On each block a solver appropriate to the block structure is used. Since the areas of complex geometry are usually small in comparison with the whole ship, it should be possible to reduce the computer storage requirements significantly.

The current version of TRANSOM includes two different solution methods which can be used on the blocks. The first is based on the program NSI2D developed for DREA under contract to the University of Toronto[3,4,5]; it is suitable for use on a single structured block. The algorithm is a variant of the pseudo-compressibility method developed by Rogers and Kwak[6]. Currently only two-dimensional flow can be modelled; NSI2D was extended to three-dimenional flow in 1996 but the changes have not yet been incorporated into TRANSOM. Turbulence may be modelled either by the Baldwin-Lomax model[7] or by the Chen-Patel variant of the k-ε model[8]. Implementation of the Baldwin-Barth model[9] is currently underway.

The second solution is based on the finite element program MEF developed at Université Laval[10]; it is suitable for use on unstructured blocks. MEF was modified at DREA to model turbulence using a variant of the k-ε model[11]. However, the turbulence modelling has not yet been incorporated into TRANSOM; the finite element solver is currently restricted to two-dimensional laminar flow.

The name TRANSOM stands for The Reynolds-Averaged Navier-Stokes Omnigenic Method, a rather strained attempt to generate an acronym that is both easy to remember and has a connection with the ship-based flows for which TRANSOM is to be used.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the overall design of TRANSOM and, in particular, the class hierarchies used to represent different types of blocks and flow solvers. The algorithms used to implement the pseudo-compressibility and finite element methods are not discussed here; two companion reports[12,13] describe these sub-solvers in detail.

TRANSOM is still under development; the current version will only solve two-dimensional flows and requires upgrading in many areas, in particular the ways in which different solution methods interact at their common block boundaries. Further development will be done both at DREA and under contract.

When TRANSOM is fully developed it will provide accurate numerical predictions of the flow into the propeller plane. It will also be applicable to many other flow problems related to ships: for example, flow around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, prediction of roll damping coefficients, and prediction of the lift and drag characteristics of lifting surfaces such as submarine control planes, rudders, and fin stabilizers.

TRANSOM is primarily concerned with three main types of object: collections of points such as grids, structured blocks, and unstructured blocks; collections of variables at the points of the grid; and solvers. Class hierarchies for each of these objects are described in Sections 2 to 4. Section 5 describes the TRANSOM main program. A user's guide for the program is also provided in Reference [14].

TRANSOM input and output files use the OFFSRF format described in Reference 15. Reference 16 describes C++ classes which implement specialized input and output streams for OFFSRF files and a base class which is inherited by all classes which read from or write to OFFSRF files; it should be read prior to this document for a proper understanding of TRANSOM input and output.

2 Grids, Blocks and Point Collections

In TRANSOM, the flow is approximated by specifying flow parameters (velocity, pressure, turbulence model variables, etc.) at each point of a grid. Borrowing from finite element jargon, we will call each point in a grid a node.

Collections of nodes may be ordered in many different ways. For example, a grid is simply a collection of sub-grids or blocks. A block may be structured, i.e. it can be ordered into a rectangular array of points, or unstructured, like the grids used in calculations using the finite element method. In TRANSOM, the organization of a collection of points is recognized as an important abstraction independent of the points themselves; it is represented by the class NodeOrg described in Section 2.3.

There are several classes for representing different types of collections of points, each of which is derived from the base class PointColl. Figure 1 shows the class hierarchy for the most important of these classes. In this section the PointColl and Grid classes are described. The class SBlock and its specializations S2dBlock and S3dBlock represent structured blocks; they are described in Reference 12. The class FEBlock represents an unstructured block organized into elements suitable for finite element calculations; it is described in Reference 13.

2.1 Nodes

The Node class is used as an index to the nodes in a grid. In TRANSOM the Node class is implemented as an integer: the number of the node.

Since a Node is simply an index, its value depends on a numbering system for the collection of nodes. In different contexts the same nodes may be numbered in different ways. For example, consider a grid of 300 nodes which are separated into three blocks of 100 nodes each. In a function which uses the whole grid, the nodes of the third block are numbered from 200 to

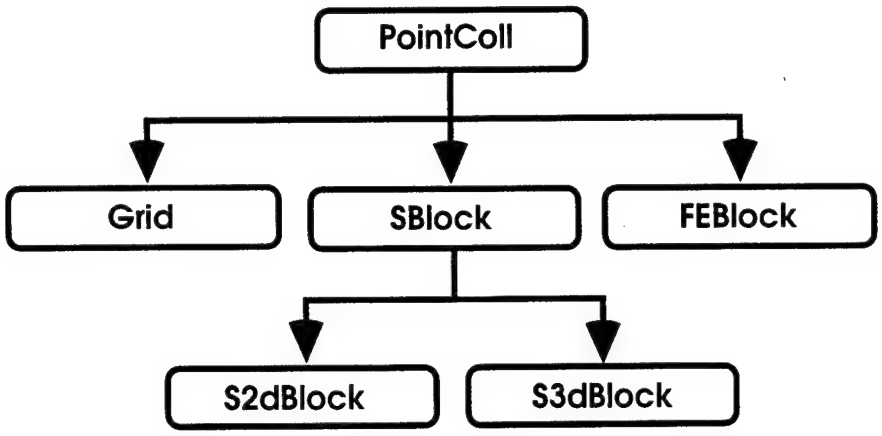

Figure 1: *Class hierarchy for the PointColl Class and its specializations*

299. However, in a function which uses only one block at a time, it may be more convenient to number the 100 nodes of the third block from 0 to 99. In both cases the Node class is used to store the number of the nodes.

The allocation of a numbering system to a collection of nodes is the job of node organizers, discussed in the Section 2.3.

2.2 Node Iterators

An iterator is a class which returns, in some well-defined sequence, each object in a collection of objects. A NodeIter is an iterator over a collection of nodes. Suppose that iter is a NodeIter. Then the iteration can be performed as follows:

```
iter.reset();
while(iter.in_range())
  {
    Node n = (Node)iter;
    // Do something with n
    iter.next();
  }
```

At any time during the iteration the current node may be obtained by casting the iterator to a Node; in the above code this is done on the first line in the while loop. This syntax is very convenient when the node is to be used as an argument to a function; then the conversion will take place automatically if the iterator is used as the argument. TRANSOM uses a similar syntax for all its iterator classes.

The member function reset() resets the iterator to the first node in the collection. The member function next() advances the iterator to the next node; most iterators also have a prev() function which backs the iteration up to the previous node. The member function in_range() returns true if the current node belongs to the collection, false if it does not (i.e. if the iteration has finished).

For maximum efficiency, the member functions of a NodeIter are not virtual functions; however, there are many cases in which it is necessary that they be so. For this reason a second class, AllNodeIter, is also provided. It is identical to NodeIter, but its member functions are declared virtual. The name AllNodeIter arises from the main use of the base class, namely to iterate over all the nodes in a collection (see the next section).

2.3 Node Organizers

Algorithms for solving the Navier-Stokes equations are almost always highly dependent on the way in which the nodes are organized. Finite difference and finite volume methods usually require structured grids: i.e. grids which are organized into rectangular arrays of nodes. For finite element methods the grids need not be structured but they must be sub-divided into elements of different kinds; each element is itself a collection of nodes. Multigrid methods can require a hierarchy of grids: each grid in the hierarchy is a subset of its parent so that a sparser covering of the fluid domain is provided. Other subsets of a grid, such as its boundaries, are also commonly considered as separate collections of nodes.

In TRANSOM the structure of collections of nodes is described by classes called node organizers. Implicit in each node organizer is a numbering scheme for the nodes. The full range of node numbers used by the numbering scheme is from zero to $N-1$, where N, is called the "maximal number of nodes". However, since the node organizer may only describe a subset of the full range of nodes, N is not necessarily the same as the number of nodes governed by the node organizer.

For example, consider a collection of eleven nodes numbered $0,\cdots,10$. A node organizer, norg, is used to represent the odd numbered nodes. Then the maximal number of nodes for norg is 11, while the number of nodes governed by norg is only 5.

Most member functions of node organizers do one of four things.

1. They return the size of the collection of nodes. All node organizers will return the total number of nodes. Node organizers for structured grids also return the dimensions of the rectangular node array. Node organizers for finite element blocks return the number of elements of which the block is comprised.
2. They return node iterators which navigate through the nodes in various ways. All node organizers return a iterator which iterates over all the nodes in the collection. Node organizers for structured blocks return iterators over a single row or column of the block.
3. They return node organizers which describe a subset of the node collection. For example, node organizers for finite element blocks will return node organizers for any of the elements which they contain.
4. They alter the node organizer so that it describes a subset (or superset) of

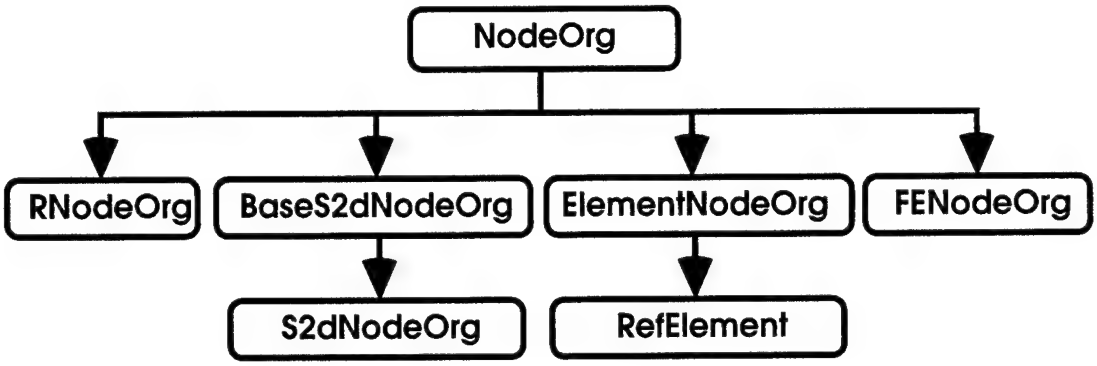

Figure 2: *Class hierarchy for the* NodeOrg *Class and its specializations*

the original collection of nodes. This type of member function is particularly useful in multigrid solvers; the member function is called to restrict the node organizer to the nodes in the next level of the hierarchy.

It is important to recognize that node organizers contain no information on the location of a node in space; that is contained in the PointColl class and its specializations, described in the next section.

The base class for node organizers is NodeOrg. It has only two member functions:

unsigned num_nodes() const;
: Returns the number of nodes organized by this NodeOrg.

virtual AllNodeIter* get_all_node_iter() const;
: Returns a pointer to an AllNodeIter which iterates over the all the nodes in the current NodeOrg. Memory for the iterator is allocated from the heap and it is the responsibility of the calling function to delete the node iterator when it is finished using it.

Figure 2 shows the class hierarchy for the most important node organizers used in TRANSOM. The RNodeOrg class defines a restriction to a subset of the nodes in a collection; it is most useful for defining the rows or columns of structured block; it is described in Reference 12. The classes Base2dNodeOrg and S2dNodeOrg are used to organize structured two-dimensional blocks; they are described in Reference 12. The FENodeOrg class organizes nodes into a collection of elements for use in finite element calculations; it is described in Reference 13. The class ElementNodeOrg is used to organize the nodes in a single finite element and its specialization RefElement provides a description of a generic reference element; both classes are described in Reference 12.

2.4 Collections of Points

The coordinate values of a collection of nodes are represented by the PointColl class which is capable of describing points in one, two, or three dimensions. It is the base of a hierarchy of classes which describe collections of points with different structures; the most important classes in the

hierarchy are shown in Figure 1.

Each PointColl contains a NodeOrg which imparts the structure to the collection of points. The prototypes for the PointColl constructors are:

PointColl(unsigned ndim, const NodeOrg &norg, double *d = 0);

Makes a PointColl whose points are of dimension ndim (one, two, or three) and are organized by norg. If d is null, the default, new memory will be allocated from the heap to store the coordinate values; this memory is freed by the PointColl destructor. Otherwise the values will be stored in the memory beginning at d; in this case the memory is not freed by the PointColl destructor. Enough memory to store ndim* N doubles is required, where N is the maximal number of nodes of the node organizer.

PointColl(const PointColl &pc);

A copy constructor; new memory is allocated and the values in pc are copied to the new PointColl.

Each PointColl also has a name by which it can be referred. For example, when reading an input file, TRANSOM uses the name to allow two different solvers to share the same block. The block is defined independently of the solvers. The solvers specify the name of the block which they will use. The block is then found and passed to the solvers for use in calculating the flow.

Values for the name and for the coordinates may be read from an OFFSRF file using an extractor overloaded for the PointColl and OFFSRF_ifstream classes (see Reference 16 for a description of OFFSRF files and the class OFFSRF_ifstream). If p is a PointColl and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> p;
```

will read the name and the coordinate values from the OFFSRF file associated with in_stream. The coordinates must be in the following format.

{NAME: *name* }
{COORDINATES

$$\begin{array}{ccc} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{array}$$

}COORDINATES

For one-dimensional point collections the y and z values are missing from the COORDINATES record. For two-dimensional point collections only the z values are missing.

All point collections have the following member functions.

unsigned num_nodes() const;

Returns the number of nodes in the point collection.

unsigned dim() const;

Returns the dimension (one, two, or three) of the points.

`const Str& get_name() const;`
Returns the name of the point collection (Str is a class representing character strings).

`NodeOrg *get_node_org() const;`
Returns a pointer to the node organizer which gives the point collection its structure.

`double* operator[](Node);`
Returns a pointer to the values of the coordinates. Thus, if p is a PointColl, then p[n][0] is the x-value of the coordinates at node n, p[n][1] is the y-value, and p[n][2] is the z-value.

2.5 Grids

A grid is a collection of points organized into blocks. It is represented by the class Grid, a specialization of the class PointColl (see Figure 1). Each of the blocks is also represented as a PointColl. Notice that, since a Grid is a type of PointColl, this means that a grid can serve as the block of a different grid.

The coordinates of the points in the grid are shared with each of the blocks. For example, suppose that g is a Grid which contains 100 points shared among three blocks. The first block uses the first 30 of these points: zero through 29. The second block uses the 50 points 30 through 79. The third block uses the 30 points 70 through 99. Notice that the ten points 70 through 79 are used by two of the blocks.

The member function get_block returns a pointer to the block specified by the argument name; its full prototype is

`PointColl* get_block(const Str &name) const;`

All the PointColl member functions described in Section 2.4 are also inherited by the Grid class.

The name, coordinate values, and the blocks of a grid are specified by a GRID record in an OFFSRF input file. The GRID record has the following format.

{GRID: *ndim npts*
{NAME: *name* }
{COORDINATES
x_1 y_1 z_1
x_2 y_2 z_2
$\vdots$ $\vdots$ $\vdots$
x_n y_n z_n
}COORDINATES
{FIRST NODE: *node block-name* }
Records to specify the blocks
}GRID

where *ndim* is the dimension of the coordinates (one, two, or three), and *npts* is the number of points in the grid. For one-dimensional point collections

the y and z values are missing from the COORDINATES record. For two-dimensional point collections only the z values are missing.

Currently three different types of blocks may be specified; each has an associated class (derived from PointColl) and OFFSRF record.

1. The record FE BLOCK causes an FEBlock to be created and added to the grid. These blocks are used in finite element solvers. The FEBlock class is described in detail in Reference 13. The FE BLOCK record contains data which define the FEBlock.
2. The record STRUCTURED BLOCK causes an S2dBlock to be created and added to the grid. This class describes structured blocks which may be used in finite difference or finite volume solvers. The S2dBlock class is described in detail in Reference 12. The STRUCTURED BLOCK record contains data which define the S2dBlock.
3. A nested GRID record causes a new Grid to be created and added as a block to the current grid.

The FIRST NODE record is used to specify which node in the grid is used for the first node in a block: *node* is the number of the node to be used as the first node in the block with name *block_name*. If no first node is specified for a given block, then the first unused node in the grid will be used. For example, consider a GRID record which specifies that a grid has three blocks each containing 30 points, but which has no FIRST NODE record. The first block will then use the grid points 0 through 29, the second will use the grid points 30 through 59, and the third the points 60 through 89.

An extractor is overloaded for the Grid and OFFSRF_ifstream classes. If g is a Grid and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> g;
```

will read the records NAME, COORDINATES, FE BLOCK, STRUCTURED BLOCK, FIRST NODE, and GRID and use them to update g.

The classes Grid, FEBlock, and S2dBlock are all able to read coordinate values from a COORDINATES record. This means that one may choose where to define the coordinate values in the input file. They can all be specified in the GRID record, as shown above. Alternatively, the points for each block may be specified within the records defining the blocks as in the following example.

```
{GRID: 2 90
  {STRUCTURED BLOCK: 2 5 10
    {NAME: Block #1 }
    {COORDINATES
      x0      y0
      ⋮       ⋮
      x49     y49
    }COORDINATES
  }STRUCTURED BLOCK
  {STRUCTURED BLOCK: 2 8 5
    {NAME: Block #2 }
```

{COORDINATES
x_{50} y_{50}
⋮ ⋮
x_{89} y_{89}
}COORDINATES
}STRUCTURED BLOCK
}GRID

If coordinate values are specified in both the grid and its blocks, the values read last (i.e. those appearing last in the file) will override the values read in earlier.

3 Flow Representation

In TRANSOM, the flow is approximated by specifying flow parameters (velocity, pressure, turbulence model variables, etc.) at each point of the grid. A hierarchy of classes is defined which allows variables of different types to be associated with nodes. A diagram of the main classes in the hierarchy is shown in Figure 3.

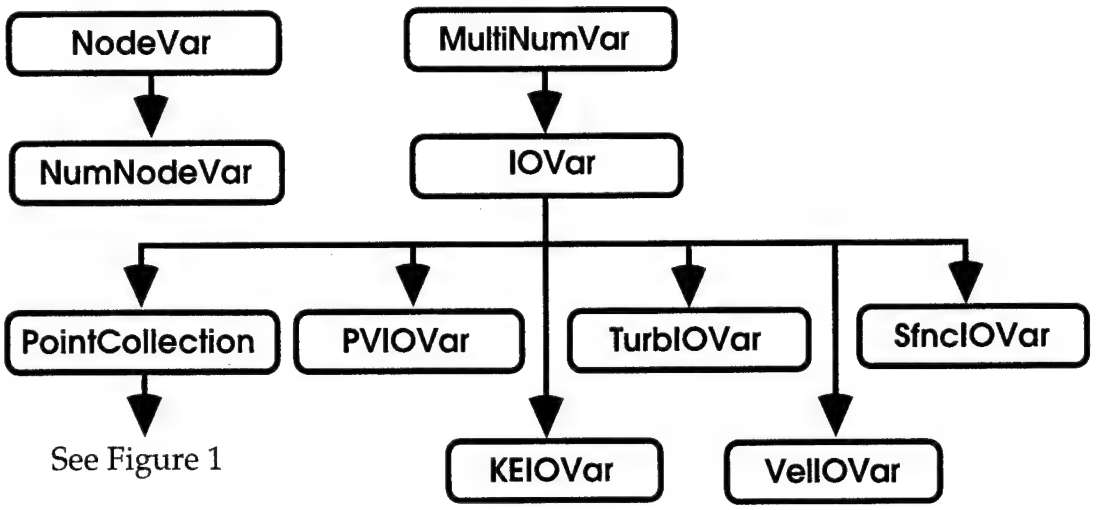

Figure 3: *Class hierarchy for classes describing flow variables*

3.1 The NodeVar class

The NodeVar class is a template used to create collections of variables each having a value at every node of a collection of points. Every instance of a NodeVar is associated with a NodeOrg; the dimensions of the NodeVar are derived from the NodeOrg and iterators returned by the NodeOrg may be used to navigate through the elements of the NodeVar.

The prototypes for the constructors of a NodeVar which stores objects of class T are:

```
NodeVar(const NodeOrg &norg, T *v = 0);
```

Creates a NodeVar which stores an object of class T at every node

governed by norg. If v is null, the default, new memory will be allocated from the heap to store the values; this memory is freed by the NodeVar destructor. Otherwise the values will be stored in the memory beginning at v; in this case the memory is not freed by the destructor. The memory required is sizeof(T)* N bytes, where N is the maximal number of nodes of the node organizer.

NodeVar(const NodeVar &nv);
A copy constructor; new memory is allocated to store the objects and their values are obtained by copying the objects in nv.

The first constructor allows values of a NodeVar to be shared with an existing NodeVar. Shared values make sub-arrays possible so that, for example, a NodeVar containing the coordinates for a block could share these coordinates with a NodeVar containing the coordinates for a whole grid.

The following member functions are defined for the NodeVar class.

unsigned num_nodes() const;
Returns the number of nodes governed by the NodeVar node organizer.

T& operator[](Node);
const T& operator[](Node) const;
The square bracket operator with Node arguments allows access to the stored values. Thus if x is a NodeVar<int>, then x[n] is the integer associated with node n.

3.2 The NumNodeVar Class

The class NumNodeVar is a specialization of the class NodeVar<double>: it associates a single floating point value with every node governed by a node organizer. Its constructors are the same as the constructors for NodeVar<double>. The NodeVar member functions described in Section 3.1 are also inherited by the NumNodeVar class.

The floating point values may be obtained using the square bracket operator with a Node argument. Thus, the following code creates a NumNodeVar containing 100 nodes, then assigns the value 1.5 at every node. Recall that the NodeIter n will automatically be converted to a Node when used as an argument to operator[](Node).

```
NodeOrg norg(100);
NumNodeVar nv(norg);
AllNodeIter *n = norg.get_all_node_iter();
while (n->in_range())
  {
    nv[n] = 1.5;
    n.next();
  }
delete n;
```

The class NumNodeVar has been given several arithmetic operators to facilitate computations which are applied to floating point values at every

node. The following member operators have been defined.

NumNodeVar& operator=(const NumNodeVar &v);
Copies the values in v.

NumNodeVar& operator-();
Negates all values.

NumNodeVar& operator+=(const NumNodeVar &v);
Increments the values by the values in v.

NumNodeVar& operator-=(const NumNodeVar &v);
Decrements the values by the values in v.

NumNodeVar& operator*=(double d);
Multiplies all values by d.

NumNodeVar& operator/=(double d);
Divides all values by d.

NumNodeVar& zero();
Sets all values to zero.

The following operators and functions are also defined.

int operator==(const NumNodeVar &vc1, const NumNodeVar &vc2);
Returns true if vc1 and vc2 are governed by the same node organizer and, for every node, the value in vc1 is the same as the value in vc2.

int operator!=(const NumNodeVar &vc1, const NumNodeVar &vc2);
Returns !(vc1 == vc2).

double abs(const NumNodeVar &vc);
Returns the square root of the sum of the squares of the values at all nodes.

double min(const NumNodeVar &vc);
Returns the minimum value stored at all nodes of vc.

double max(const NumNodeVar &vc);
Returns the maximum value stored at all nodes of vc.

double min_abs(const NumNodeVar &vc);
Returns the value stored in vc whose absolute value is smallest.

double max_abs(const NumNodeVar &vc);
Returns the value stored in vc whose absolute value is largest.

3.3 The MultiNumVar Class

The class MultiNumVar is similar to the NumNodeVar class: it associates several floating point values with every node. The MultiNumVar class has three constructors with the following prototypes.

MultiNumVar(unsigned nd, const NodeOrg &norg, double *d = 0);
Constructs an MultiNumVar governed by the NodeOrg norg. At each node, nd doubles are stored. The doubles are stored starting at memory location d. If d is null, new memory will be allocated for the stored values.

MultiNumVar(const NodeOrg &norg, const MultiNumVar &iv, const Node &n);
Shared Copy constructor: constructs an MultiNumVar governed by norg and with the same number of doubles as iv. The values in iv starting at node n are shared.

MultiNumVar(const MultiNumVar &iv);
A copy constructor: new memory is allocated for the stored values.

The following member functions are also defined.

unsigned num_nodes() const;
Returns the number of nodes at which the values are stored.

unsigned dim() const;
Returns the number of values stored at each node.

Given a Node argument, the square bracket operator returns a pointer to the doubles associated with the node. Thus, the following code creates a MultiNumVar with 100 nodes and associates 10 doubles with each node. At each node the values of the doubles are set to 1.0 to 10.0.

```
NodeOrg norg(100);
MultiNumVar nv(10,&norg);
AllNodeIter *n = norg.get_all_node_iter();
while (n->in_range())
  {
    for (unsigned i = 0; i < nv.dim(); i++)
      nv[n][i] = i + 1;
    n.next();
  }
delete n;
```

The MultiNumVar class has the same arithmetic operators as the class NumNodeVar, but they operate on all the doubles stored at each node.

3.4 The IOVar Class

The IOVar class is a specialization of the MultiNumVar class which is used to represent global variables in TRANSOM. It provides a means of organizing the floating point values at each node so that they can be associated more easily with independent variables. A subset of the floating point values can be identified and the arithmetic operators restricted so that they act upon that subset only. Input and output functions are also defined so that the values can be read from or written into an OFFSRF file.

The organization of the floating point values is specified using successive calls to the IOVar member function add_to_structure whose full prototype is

void add_to_structure(const Str &s, unsigned n, unsigned off);

A call to add_to_structure assigns the name s to the n floating point values beginning at the offset off. For example, suppose the IOVar pv is intended to store the values of pressure and velocity in a two-dimensional flow. Three floating point values are required at each node and it is decided that the first should be pressure followed by the two velocity components. The IOVar can

be set up in this manner using the following code.

```
IOVar pv(3,norg);  // norg is a pointer to a NodeOrg
pv.add_to_structure("Pressure", 1, 0);
pv.add_to_structure("Velocity", 2, 1);
```

The value of the pressure at node n may then be obtained by pv[n][0] and the two velocity components by pv[n][1] and pv[n][2].

The member functions clear_active and activate_only may also be used to change the organization of the variables. Their full prototypes are

```
void clear_active();
void activate_only(const Array<Str> &fields);
```

The former removes all organization of the variables; a new organization may be defined using successive calls to add_to_structure.

The member function activate_only is used to restrict the organization of an IOVar to certain variables. The argument fields is a list of names of the variables which are to remain active. If a variable name is not in the list, it will no longer be active. If fields contains a name which is not already part of the IOVar organization, a fatal error will result.

The values for any of the named variables may be read from an OFFSRF file using an extractor overloaded for the IOVar and OFFSRF_ifstream classes (see Reference 16 for a description of OFFSRF files and the class OFFSRF_ifstream). If pv is the IOVar defined above and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> pv;
```

will read the values for the pressure and velocity at each node from the OFFSRF file associated with in_stream. The OFFSRF records should be named "Pressure" and "Velocity", the same names used when defining the structure of the floating point values with the calls to add_to_structure.

{Pressure
p_1 p_2 ... p_n
}Pressure
{Velocity
v_{x0} v_{y0}
v_{x1} v_{y1}
⋮ ⋮
v_{xn} v_{yn}
}Velocity

If one of the records is missing, the values for that variable will remain undefined.

Similarly, an inserter is overloaded for the IOVar and OFFSRF_ofstream classes. If pv is the IOVar defined above and out_stream is an OFFSRF_ofstream, then the code

```
out_stream << pv;
```

will write "Pressure" and "Velocity" records in the OFFSRF file associated with in_stream.

The IOVar constructors are similar to those of the MultiNumVar class. Their prototypes are:

```
IOVar(unsigned nd, const NodeOrg &norg, double *d = 0);
IOVar(const NodeOrg &norg, const IOVar &iv, const Node &n);
IOVar(const IOVar &iv);
```

An important function of the IOVar class is to restrict a set of global variables to a subset of independent variables used by a solver. For example, consider a solver, S, which solves the laminar Navier Stokes equations on a single block. The solver is split into two sub-solvers, S_1 and S_2, which are solved in sequence: S_1 first determines the values of the pressure and velocity, then S_2 determines the streamfunction. A member of the main solver is an IOVar, vars, which contains the fields "Pressure", "Velocity", and "Streamfunction"; it is passed to both S_1 and S_2. Within S_1 the streamfunction is not used. Therefore, a new IOVar, pv, is constructed from vars; it shares the values in vars but restricts the active fields to "Pressure" and "Velocity". This is done as follows (a simpler method using the specialized class PVIOVar is described in Section 3.5.2).

```
IOVar pv(*vars->get_node_org(),vars,0);
Array<Str> fields(2);
fields[0] = "Pressure";
fields[1] = "Velocity";
pv.activate_only(fields);
```

Similarly, in S_2, an IOVar restricted only to the "Streamfunction" field is used. Notice that, since the values of the restricted IOVar are shared with vars, the main solver, S, has access to all the variables and can be used to handle all the input and output in a consistent way.

3.5 Flow Representation Classes

The classes PVIOVar, VelIOVar, SfncIOVar, TurbIOVar, and KEIOVar are specializations of the IOVar class designed specifically for representing flow variables.

3.5.1 The VelIOVar Class

A VelIOVar is an IOVar containing a single active field named "Velocity". It has three constructors which correspond closely with the IOVar constructors.

VelIOVar(unsigned nd, const NodeOrg &norg, double *d = 0);
: Creates a VelIOVar for velocity variables of dimension nd. If d is non-null, the velocity values will be stored starting at memory location d; otherwise new memory will be allocated (and destroyed when the VelIOVar is destroyed).

VelIOVar(const NodeOrg &norg, const IOVar &iv, const Node &n);
: Creates a VelIOVar which shares the velocity values with the IOVar iv starting at node n. If iv does not contain a field named "Velocity" whose length is nd, a fatal error will result.

VellOVar(const VellOVar &iv);

A copy constructor; new memory is allocated for the velocity values and all values are copied to the newly constructed VellOVar.

The member function vel allows easy access to the velocity values; given a Node argument, it returns a pointer to the velocity values at that node. Thus, if v is a VellOVar, then v.vel(n)[0] is the x-component of the velocity at node n.

3.5.2 The PVIOVar Class

A PVIOVar is an IOVar containing two active fields named "Pressure" and "Velocity". It has the following three constructors.

PVIOVar(unsigned nv, const NodeOrg &norg, double *d = 0);

Creates a PVIOVar with velocity variables of dimension nd. If d is non-null, the pressure and velocity values will be stored starting at memory location d; otherwise new memory will be allocated (and destroyed when the PVIOVar is destroyed).

PVIOVar(const NodeOrg &norg, const IOVar &iv, const Node &n);

Creates a PVIOVar which shares its values with the IOVar iv starting at node n. If iv does not contain a field named "Pressure" and a field named "Velocity" whose length is nd, a fatal error will result.

PVIOVar(const PVIOVar &iv);

A copy constructor; new memory is allocated for the velocity values and all values are copied to the newly constructed PVIOVar.

The member functions p and vel allow easy access to the velocity values. The latter is the same as the vel function of the VellOVar class. Given a Node argument, the member function p returns the pressure value at that node. Thus, if pv is a PVIOVar, then pv.p(n) is the pressure at node n.

3.5.3 The SfnclOVar Class

A SfnclOVar is an IOVar containing a single active field named "Streamfunction". It has the following three constructors.

SfnclOVar(const NodeOrg &norg, double *d = 0);

Creates a SfnclOVar which stores only the value of the streamfunction at each node. If d is non-null, the streamfunction values will be stored starting at memory location d; otherwise new memory will be allocated (and destroyed when the SfnclOVar is destroyed).

SfnclOVar(const NodeOrg &norg, const IOVar &iv, const Node &n);

Creates a SfnclOVar which shares its values with the IOVar iv starting at node n. If iv does not contain a field named "Streamfunction", a fatal error will result.

SfnclOVar(const SfnclOVar &iv);

A copy constructor; new memory is allocated for the velocity values and all values are copied to the newly constructed SfnclOVar.

The member function sfnc allows easy access to the streamfunction val-

ues; given a Node argument, it returns the value of the streamfunction at that node. Thus, if s is a SfncIOVar, then s.sfnc(n) is the streamfunction at node n.

3.5.4 The TurbIOVar Class

A TurbIOVar is an IOVar containing a single active field named "Turbulent Viscosity". It has three constructors which are similar in function to those of the SfncIOVar class.

```
TurbIOVar(const NodeOrg &norg, double *d = 0);
TurbIOVar(const NodeOrg &norg, const IOVar &iv, const Node &n);
TurbIOVar(const TurbIOVar &iv);
```

The member function visc allows easy access to the values of turbulent viscosity.

3.4.5 The KEIOVar Class

A KEIOVar is an IOVar containing two active fields named "k" and "Epsilon"; it stores value of k and ε used by the k-ε turbulence model and its many variants. It has three constructors which are similar in function to those of the SfncIOVar class.

```
KEIOVar(const NodeOrg &norg, double *d = 0);
KEIOVar(const NodeOrg &norg, const IOVar &iv, const Node &n);
KEIOVar(const PVIOVar &iv);
```

The member functions k and eps allows easy access to the values of k and ε respectively.

4 Solvers

A solver is an abstraction of iterative methods used to determine a solution to some problem. Let x be the state vector for the problem. The solution is found by postulating an initial state, x_0, then determining successive values of the state by applying a function, F, which will be called the sweep.

$$x_{n+1} = F(x_n)$$

The iteration must be stopped at some point. For this purpose a stopping test is defined (we avoid the term convergence criterion because, in many cases, it will be desirable to stop the iteration before convergence is attained); it is a Boolean-valued function of the state vector, $C(x)$. The solver algorithm is then:

$$\begin{aligned} S(x) = \{\ & x \leftarrow x_0 \\ & \text{while not } C(x) : \{\ x \leftarrow F(x)\ \} \\ \}\ & \end{aligned} \tag{1}$$

Sometimes it will be useful to define new sweeps using existing sweeps as building blocks: for example, the sweep for a multigrid solver is defined as a sequence of sweeps on each of the grids. Since this implies that a sweep

might be used in more than one solver, it is best to define the solver and the sweep as separate classes. Similarly, it may be useful to define different stopping tests for a solver; moreover, the stopping test for a sweep may be different depending upon its use within different solvers. Hence, the stopping test should **not** be implemented as a member function of either the solver class or the sweep class; it is implemented as a separate class.

Since, in general, both the sweep and the stopping test will need to use the state vector, it must be defined separately and shared.

4.1 The Sweep class

The Sweep class is defined to have three member functions:

virtual void initialize():
: Initializes the state vector to x_0 and performs any other initialization necessary for the efficient execution of the iteration function $F(x)$.

virtual void sweep():
: Implements the iteration function $F(x)$ by altering the state vector from the value x_n to x_{n+1}.

virtual void finalize():
: Performs any tasks necessary after the iteration has been stopped: for example, freeing of memory allocated by initialize(). Often finalize() does nothing.

The Sweep class is derived from the class OFFSRF described in Reference 16. Hence, every specialization of Sweep has an inserter and extractor defined for the OFFSRF streams OFFRSF_ofstream and OFFSRF_ifstream. The inserter and extractor for the class Sweep itself do nothing.

The most important specializations of the Sweep class are shown in the class hierarchy of Figure 4. The ResidualSweep class is described in Section 4.6. Its specializations, SPC2dPVSweep and SPC2dKESweep, along with SPC2dBLSweep are used to implement a two-dimensional pseudo-compressibility solution of the Navier-Stokes equations on a structured grid. An SPC2dPVSweep calculates pressure and velocity in the flow, an SPC2dKESweep calculates the turbulence variables using the k-ε turbulence model, and SPC2dBLSweep calculates the turbulent viscosity using the Baldwin-Lomax turbulence model. The class SPCMtxSweep solves the linear system which arises in the pseudo-compressibility sweeps; a line relaxation method is used. These classes are described in detail in Reference 12. The SeqSweep and ConcSweep classes are described in Section 4.4. The FESweep class is a generic finite element sweep; it assembles a linear system of equations and solves it. Its specialization FE2dNSSweep is used to solve the two-dimensional Navier-Stokes equations. The FESfncSweep calculates the streamfunction for an incompressible velocity field. The classes FESweep, FE2dNSSweep and FESfncSweep are all described in Reference 13.

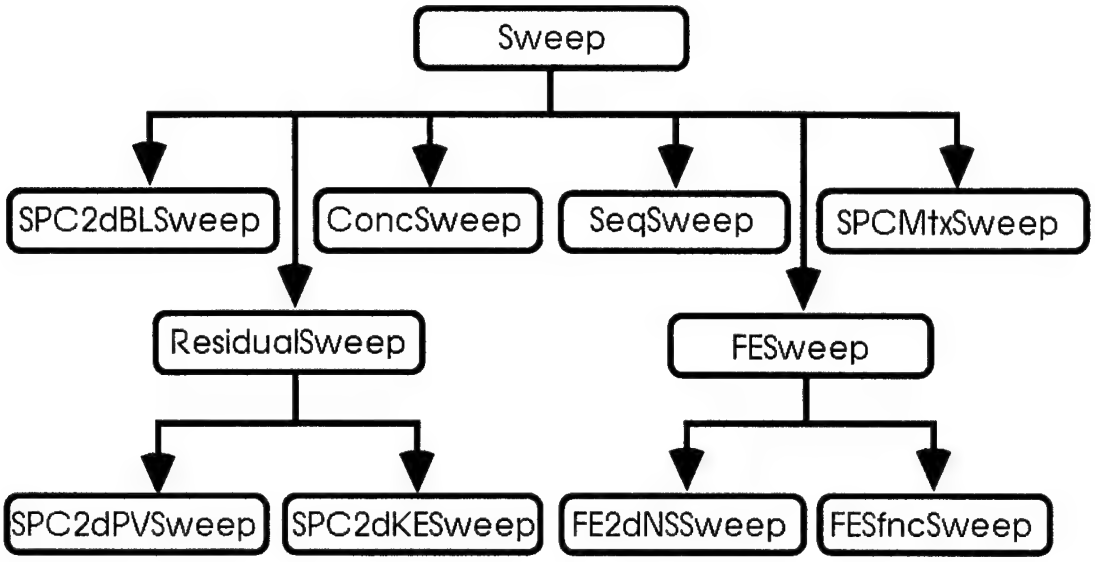

Figure 4: *Class hierarchy for* Sweep *and its specializations*

4.2 The StopTest class

The StopTest class has two member functions:

virtual void initialize();
: Performs any initialization necessary for the proper functioning of the stop test.

virtual int stop();
: Returns true if the sweeps should be stopped; false if they should continue. Thus, stop() implements the function $C(x)$. Note that, as described above, the state vector x is shared data of both the StopTest class and the Sweep class; hence, it is not provided as an argument to stop(). Of course, this also circumvents the problem that the type of data is not yet specified so that, for a strongly typed language like C++, the type of the argument would not be known.

There are several simple but important specializations of stop tests. The TrueStopTest always returns true; the FalseStopTest always returns false. The OrStopTest combines two stop tests, C_1 and C_2, to obtain $C = C_1 \wedge C_2$. An AndStopTest is similar but is defined by $C = C_1 \vee C_2$.

Figure 5 shows the hierarchy of classes derived from StopTest. A ReportStopTest is a stop test which reports on its state whenever is stop() function is executed; it is described in Section 4.2.1. An NTimesStopTest allows the sweep to be performed n times; it is described in Section 4.2.2. A CompDataStopTest returns true when successive values of the state vector have changed by less than some pre-defined small amount; it is described in Section 4.2.3. A ResidualStopTest is a stop test suitable for a residual solver; it is described in Section 4.6.1. The SeqStopTest and ConcStopTest classes represent stop tests suitable for solvers which use more than one sweep; they are described in Sections 4.4.1 to 4.4.3.

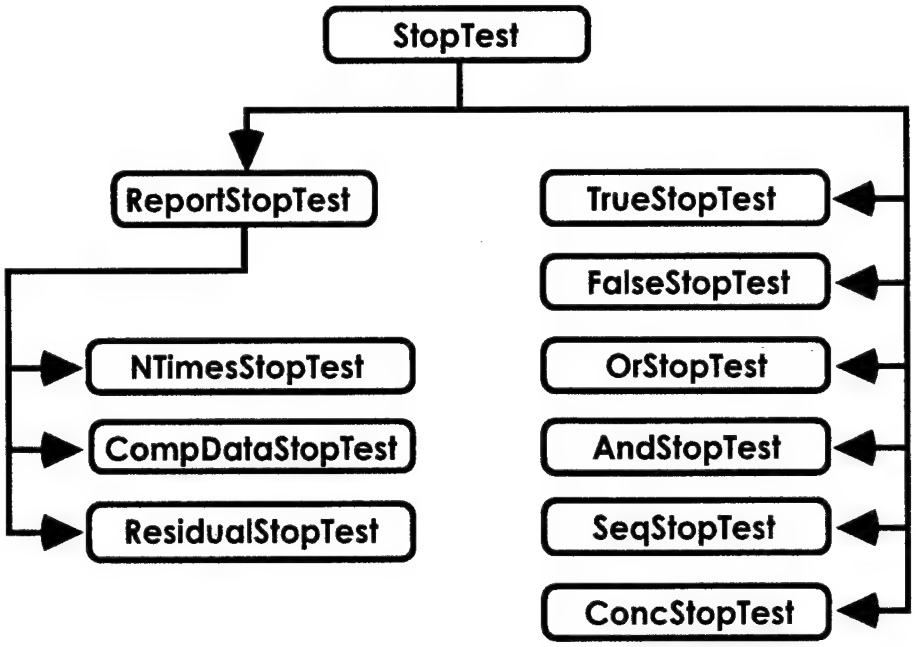

Figure 5: *Class hierarchy for StopTest and its specializations*

4.2.1 The ReportStopTest Class

A ReportStopTest is a StopTest with the ability to report on its state whenever the stop function is called. The virtual function report is used to write the report. A character string may be prepended to the report; normally this is used to identify the solver to which the stop test applies.

Each ReportStopTest can be set so that the report is written before the stop test is executed, after the stop test is executed, or not at all. The member function set_report() is used for this purpose. Its argument should be one of the following:

ReportStopTest::before
: Write the report before the stop test is executed.

ReportStopTest::after
: Write the report after the stop test is executed, but only if the stop test returned true.

ReportStopTest::after_always
: Write the report after the stop test is executed, irrespective of the result of the stop test.

ReportStopTest::none
: Do not write the report.

For example, if rst is a ReportStopTest, then

```
rst.set_report(ReportStopTest::before);
```

sets the stop test so that the report will be written before the stop test is executed. By default, a ReportStopTest is set so that no report is written.

The ReportStopTest class does not define the initialize() virtual function inherited from StopTest. It must be defined by specializations of the class.

The constructors and member functions are:

ReportStopTest();
Creates a new ReportStopTest. By default no report will be written.

void report(ostream&);
Writes a report on the state of the stop test to the output stream.

virtual void report_no_prepend(ostream&) = 0;
Writes the report with no prepended string. This function must be defined by specializations of the class.

virtual int stop();
Calls the function stop_no_report() to execute the stop test. The report is written before or after this call as specified by the most recent call to set_report().

virtual int stop_no_report() = 0;
Executes the stop test with no reporting. This function must be defined by specializations of the class.

void set_report_string(const Str &s)
Sets the character string which is prepended to the report.

4.2.2 The NTimesStopTest class

An NTimesStopTest allows the sweep to be performed n times; its stop() function returns true the $(n+1)^{st}$ time it is called. It is a specialization of the ReportStopTest class. Its constructor has the following prototype:

```
NTimesStopTest(unsigned m);
```

where m is the number of times the sweep is to be executed before the stop test returns true. The initialize() function resets the iteration count so that the stop test will be executed m more times. The number of times to execute the iteration may be changed using the following member function.

```
void set_number(unsigned m);
```

The NTimesStopTest inherits the member functions set_report(), report() and set_report_string(), from ReportStopTest. The report from the NTimesStopTest simply prints the current iteration count followed by an end-of-line character. Thus, the code

```
NTimesStopTest nst(3);
nst.set_report(ReportStopTest::before);
nst.set_report_string("Iteration ");
while (nst.stop());
```

will cause the following to be written to cout:

```
Iteration 0
Iteration 1
Iteration 2
```

If ReportStopTest::after_always had been used, then the result would have been

```
Iteration 1
Iteration 2
Iteration 3
```

since in this case the iteration count is incremented prior to the report.

4.2.3 The CompDataStopTest class

Another important type of stop test checks convergence by comparing successive values of the state vector; if it has not changed much, then the solver has converged. The CompDataStopTest is used to implement this condition. In a CompDataStopTest the state vector is represented by an IOVar. When the CompDataStopTest is constructed, a copy of the state vector is made; it is used to store the state vector from the previous iteration. The stop test always returns false the first time it is called after initialization. Otherwise it compares the current state vector with the state vector from the previous iteration and returns true if one of two conditions is satisfied.

1. The absolute value of the largest difference in the state vectors is less than the small pre-defined value ε_{acc}. This condition stops the solver when the solution has converged.
2. The absolute value of the largest difference in the state vectors is greater than the large pre-defined value ε_{div}. This condition stops the solver when the solution has diverged.

The CompDataStopTest constructors and member functions are:

```
CompDataStopTest(IOVar *d, double a, double maxa);
```
Makes a CompDataStopTest with state vector obtained from d, with ε_{acc} set to a, and with ε_{div} set to maxa.

```
virtual void initialize();
```
Initializes the stop test. The next call to stop() will return false.

```
virtual int stop();
```
Executes the stop test.

```
void set_max_residual(double maxa);
```
Sets the value of ε_{div} to maxa.

```
void set_acc(double a);
```
Sets the value of ε_{acc} to a.

The CompDataStopTest class is a specialization of the ReportStopTest class. The member functions set_report(), report(), and set_report_string() are inherited from ReportStopTest. The report from the CompDataStopTest prints the maximum difference found for each of the active variables in the IOVar d. The default string prepended to the report is "Max.Acc. = ", but it can be changed using set_report_string(). Thus, if cdst is a CompDataStopTest constructed using a PVIOVar (three active variables: pressure and two velocity components), then the report is similar to the following.

```
Max.Acc. =      0.123456    -0.0256543     0.0365476
```

The first number is the maximum difference in the pressure values, and the latter two numbers are the maximum differences in each of the velocity components.

4.3 The Solver class

The Solver class is an implementation of the abstract solver defined above. It has a shared Sweep, sw, and a shared StopTest, st, as data and the following member functions.

virtual void initialize():
Initializes the sweep sw and the stop test st.

virtual void solve():
Implements the solution algorithm of equation (1).

virtual void finalize():
Finalizes the sweep sw and the stop test st.

The Solver class has two constructors.

Solver(Sweep *s, StopTest *t);
Makes a Solver with sweep sw and stopping test st.

Solver(const Solver&);
Copy constructor.

The Solver class is derived from the class OFFSRF described in Reference 16. Hence, every specialization of Solver has an inserter and extractor defined for the OFFSRF streams OFFRSF_ofstream and OFFSRF_ifstream. The Solver inserter simply calls the inserter for its sweep. Similarly, its extractor calls the extractor for its sweep.

The most important specializations of the Solver class are shown in the class hierarchy of Figure 6. The ResidualSolver class is described in Section 4.6. The ConnectedSolver class is described in Section 4.4.3.2. The FE2dNSSolver class solves the laminar two-dimensional Navier-Stokes equations using the finite element method; it is described in Reference 13. The SeqSolver and ConcSolver classes are used to combine several different solvers so that they may be executed as a single solver; they are described in Section 4.4. The SeqFlowSolver and ConcFlowSolver classes are specializations of SeqSolver and ConcSolver which are specifically designed for use by flow solvers; they are described in Section 4.5. An SPC2dSolver solves the two-dimensional Navier-Stokes equations using a pseudo-compressibility method; it is described in Reference 12. Finally, an SPC2dTurbSolver solves the two-dimensional turbulent transport equations on a structured grid; it is used in conjunction with the SPC2dSolver; it is described in Reference 12.

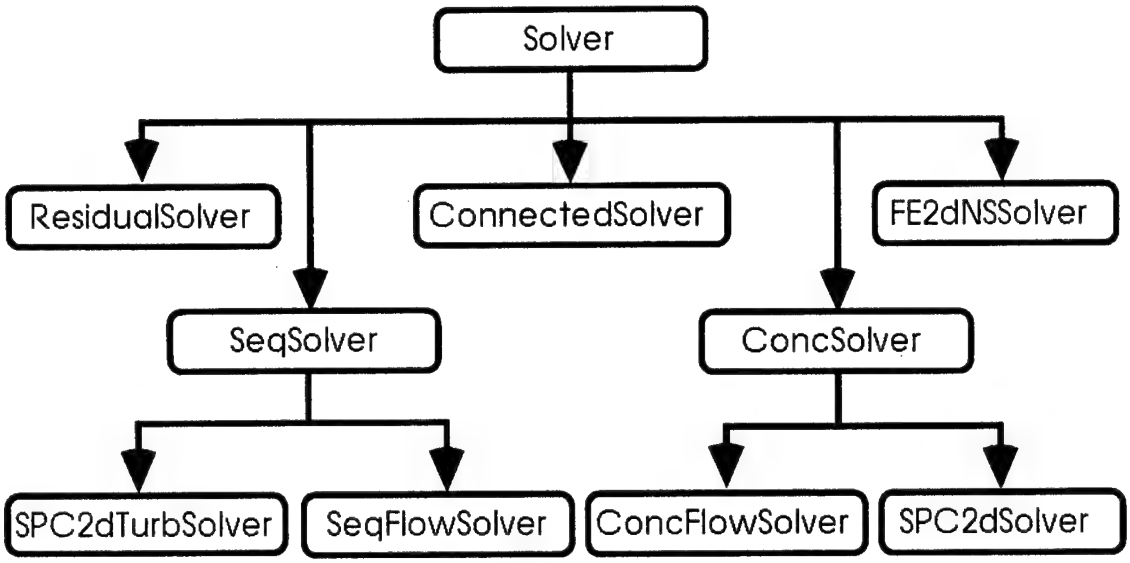

Figure 6: *Class hierarchy for Solver and its specializations*

4.4 Multiple Sweep Solvers

An important specialization of a solver is one which consists of a collection of sub-solvers each applied to a subset of the state vector. For $n = 1,...,N$, let $S_n(x_n)$ be the collection of sub-solvers each with its sweep, F_n, and stop test C_n. Each of these solvers can be run independently to achieve a solution over its sub-state-vector, x_n. However, in general, the sub-state-vectors will not be disjoint so that, when the solvers are run sequentially, they interact, the solution algorithm of one solver affecting the solution of another.

Moreover, the sub-state-vectors may be implemented in such a way that a single parameter has copies in more than one sub-state-vector. In this case whenever the parameter is changed in one sub-state-vector, its value must also be altered in all the others in which it appears. Often, but not always, such redundancy will be avoided by sharing state variables among the solvers.

It is probably clear that, in presenting the multiple sweep solver, we have in mind a multi-block Navier-Stokes solver. On each block a solution method is defined. Boundaries may be shared by two blocks and each of the block solvers may store copies of the independent variables on the boundaries.

There are two main types of multiple sweep solvers; they are described in the following two sections.

4.4.1 Concurrent Solvers

A concurrent solver is a multiple sweep solver which finds a solution to all the sub-solvers concurrently. Each of the sweeps is executed in turn

(perhaps more than once), the cycle of sweeps continuing until the stop tests of all sub-solvers return true. More precisely:

1. The stop test for the solver is the union of all the sub-stop-tests: i.e. the stop test returns true when all of the sub-stop-tests return true.
   $C(x_1,...,x_N) = C_1(x_1) \vee ... \vee C_N(x_N)$
2. The solver is initialized by initializing all sub-sweeps and sub-stop-tests.
   for $n = 1$ to N { initialize $C_n(x_n)$ and $F_n(x_n)$ }
3. The algorithm for the sweep for the concurrent solver is:

```
for n = 1 to N
 { initialize C*_n(x_n)
   while C*_n(x_n) : x_n ← F_n(x_n)
   for m = 1 to N & m ≠ n :
    { if (x_n and x_m are redundant) T_nm(S_n(x_n), S_m(x_m)) }
 }
```

   (2)

   where the C_n^*. are a new stop tests and T_{nm} is a function which maintains the consistency of the sub-state-vectors x_n and x_m by changing the values of x_m according to the values of x_n.
4. The solver is finalized by finalizing all sub-sweeps.
   for $n = 1$ to N { finalize $F_n(x_n)$ }

Notice that the arguments of T_{nm} are not the state vectors x_n and x_m, but the solvers $S_n(x_n)$ and $S_m(x_m)$. This is because the algorithm to update the state vectors will usually depend on the types of solvers used and not only on the data itself.

The sweep algorithm can be made more efficient if it is assumed that the redundancy of all pairs (x_n, x_m) is known *a priori*. In that case the loop over m can be restricted to the sub-state-vectors which are known to be redundant with x_n. In many cases there will be no redundant state vector pairs.

4.4.2 Sequential Solvers

A sequential solver consists of a sequence of solvers; when the stop test of one solver is satisfied, the next solver in the sequence is started. As with concurrent solvers, the sub-solvers may share portions of the state vector; hence, after each sub-solver has been solved, update functions T_{nm} must be called to preserve the consistency of the state vectors.

A solver which finds a solution to all the sub-solvers in sequence may be defined as follows. For $n = 1,...,N$, let $S_n(x_n)$ be the collection of sub-solvers each with its sweep, F_n, and stop test C_n. At any time only one of the solvers,

the n^{th}, is active.

1. The stop test tests the stop test of the current solver. If it is true, the current solver is finalized, its connections updated, then the next solver in the sequence is activated and initialized. The stop test returns true when the stop test of the last solver in the sequence returns true.

$$
\begin{aligned}
&C(x_1,\ldots,x_N) = \\
&\{\ \text{while } C_n(x_n)\ \&\ n < N \\
&\quad \{\ \text{initialize } S_{n+1}(x_{n+1}) \\
&\qquad \text{for } m = 1 \text{ to } N\ \&\ m \neq n: \\
&\qquad\quad \{\ \text{if } (x_n \text{ and } x_m \text{ are redundant}) \ T_{nm}(S_n(x_n), S_m(x_m))\ \} \\
&\qquad \text{finalize } S_n(x_n) \\
&\qquad n \leftarrow n+1 \\
&\quad \} \\
&\quad \text{return } C_n(x_n) \\
&\}
\end{aligned}
\tag{3}
$$

2. The solver is initialized by resetting n to 1 and initializing $S_1(x_1)$.
3. The sweep simply executes the sweep of the current solver:
   $F(x_1,\ldots,x_N) = F_n(x_n)$
4. The solver finalize function does nothing, since all the sub-solvers are finalized during the execution of the stop test.

4.4.3 Implementation of Multiple Sweep Solvers

In TRANSOM, both concurrent and sequential solvers are implemented with the help of several special classes.

4.4.3.1 The SweepConnection Class

The SweepConnection class represents the connection function $T(S_1,S_2)$ between two solvers having redundant state vectors. It has two member functions:

virtual void initialize():
: Performs initializations required for the efficient execution of $T(S_1,S_2)$

virtual void update():
: Implements the function $T(S_1,S_2)$. The solver arguments S_1 and S_2 will normally be specified as arguments to the constructor when an instance of SweepConnection is created.

4.4.3.2 The ConnectedSolver Class

To implement the sweep algorithms of the multiple sweep solvers the class ConnectedSolver is defined. A ConnectedSolver, S, consists of a solver and a list of connection functions, $T(S,S_n)$.

It is important that it be possible to promote any Solver to a ConnectedSolver; hence it should have a constructor with the following prototype.

```
ConnectedSolver(Solver &s);
```

When this constructor is used, the ConnectedSolver should behave exactly as did the original solver, s, except that the ability to add and to execute connection functions is provided. A natural way to implement this constructor would be to copy the sweep and stop test from s; however, in that case, any overloaded member functions of s would be lost. Instead the ConnectedSolver class stores a pointer, slv, to the original solver; the Solver member functions are then overloaded so that they execute the corresponding member function of slv.

Notice that in this implementation a ConnectedSolver stores two sweeps, Solver::sw and slv->sw, and two stop tests, Solver::st and slv->st. The duplication of sweeps has no utility; Solver::sw is always ignored. However, the second stop test can be used to store the stop test, C_n^*, required by the concurrent solver. Accordingly, constructors are supplied to allow Solver::st and slv->st to be defined independently.

When a connection between two solvers is defined in an input file, there must be some means of identifying the solvers which are to be connected. To make this easier, a connected solver also has a name. ConnectedSolver overloads the Solver extractor so that it may read the name from a NAME record having the following format.

{NAME: *name* }

where *name* is any character string. When comparing the name with other names, leading and trailing whitespace is ignored but the comparison is case sensitive. Other input records are read by calling slv->read_record (see Reference 16).

Following are prototypes for the ConnectedSolver constructors and member functions.

```
ConnectedSolver(Solver &s);
```
Creates a ConnectedSolver from the solver s. The pointer slv is set to &s. The list of connection functions is empty; they must be added using the member function insert.

```
ConnectedSolver(const ConnectedSolver &cs, StopTest *t = 0);
```
Copies cs including its list of connection functions. If t is non-null, the stop test pointer Solver::st will be set to t; otherwise it will be set to slv->st.

```
void insert(SweepConnection *sc);
```
Adds a connection function to the list

```
virtual void initialize();
```
Initializes the ConnectedSolver prior to its solution. Initializes the solver slv, the stop test Solver::st, and all of the connection functions.

virtual void finalize();
Finalizes the ConnectedSolver after its solution; equivalent to slv->finalize().

virtual void update_connections();
Calls update() for every connection function in the list.

Solver* get_solver() const;
Returns slv.

StopTest* get_original_stop_test() const;
Returns a pointer to the stop test used by slv: i.e. returns slv->st.

const Str& get_name() const;
Returns the solver name.

4.4.3.3 The ConcSolver Class

Concurrent solvers are represented by the class ConcSolver; the sub-solvers are represented by a list of ConnectedSolvers. For each sub-solver, S_n, the stop test C_n is represented by the stop test slv->st in the ConnectedSolver; the stop test C_n^* is represented by Solver::st. The stop test for the ConcSolver is a ConcStopTest; its stop() function implements algorithm described in Section 4.4.1.

The ConcSolver class has the following constructors and member functions.

ConcSolver():
Creates a new ConcSolver with an empty list of sub-solvers. Sub-solvers must be added using one of the insert functions defined below. During the sweep, the sub-solvers will be executed in the order in which they are added to the list.

void insert(ConnectedSolver *s, StopTest *t = 0);
Adds a new connected solver to the list of sub-solvers. If t is non-null, the stop test cs->st is replaced by t: i.e. when the sweep is performed, the stop test C^* will be evaluated using t.

void insert(Solver *s, StopTest *t);
Add a new solver to the list of sub-solvers by first converting it to a ConnectedSolver with an empty list of connection functions. The stop test C is represented by s->st and the stop test C^* by t.

int is_empty() const;
Returns true if there are no sub-solvers.

virtual void initialize();
Initializes all the sub-solvers prior to execution of the sweeps.

virtual void sweep();
Executes the sweeps as described in Section 4.4.1.

virtual void finalize();
Finalizes all sub-solvers after execution of the sweeps.

4.4.3.4 The SeqSolver Class

Sequential solvers are represented by the class SeqSolver; the sub-solvers are represented by a list of ConnectedSolvers. Since only one stop test is required for each sub-solver, in the ConnectedSolvers the stop tests slv->st and Solver::st are always the same. The stop test for the SeqSolver is a SeqStopTest; its stop() function implements algorithm (3).

The SeqSolver class has the following constructors and member functions.

SeqSolver();
: Makes a SeqSolver with an empty list of sub-solvers. The sub-solvers must be added using the insert function.

void insert(ConnectedSolver *s);
: Adds a new connected solver to the list of sub-solvers.

void insert(Solver *s);
: Adds a new solver to the list of sub-solvers by first converting it to a ConnectedSolver with an empty list of connection functions.

virtual void initialize();
: Initializes the first sub-solver prior to execution of the sweeps.

virtual void sweep();
: Executes the sweep algorithm described in Section 4.4.2.

virtual void finalize();
: Does nothing, since all sub-solvers are finalized during execution of the stop tests.

4.5 Flow Solvers

The FlowSolver class is an abstract base class for solvers used to calculate fluid flow. Its state consists of a Grid, an IOVar used to represent flow variables having values at all nodes, and a single double used to store the Reynolds number. The prototypes for the FlowSolver constructors are:

FlowSolver(Grid *g = 0, double r = -1.0, IOVar *v = 0);
: Creates a FlowSolver using the grid g, the flow variables v, and the Reynolds number r. If the grid is null, it must be defined by reading it from an input file using the extractor described below. A negative Reynolds number is used as a flag that the Reynolds number has not been defined.

FlowSolver(const FlowSolver&);
: A copy constructor.

The FlowSolver class has two specializations: SeqFlowSolver and ConcFlowSolver. Each adds a list of sub-solvers to the FlowSolver: SeqFlowSolver solves the sub-solvers in sequence, ConcFlowSolver concurrently. This is implemented by deriving the former from the SeqSolver class as well as FlowSolver, the latter from the ConcFlowSolver class as well as FlowSolver. This multiple inheritance hierarchy is illustrated in Figure 7.

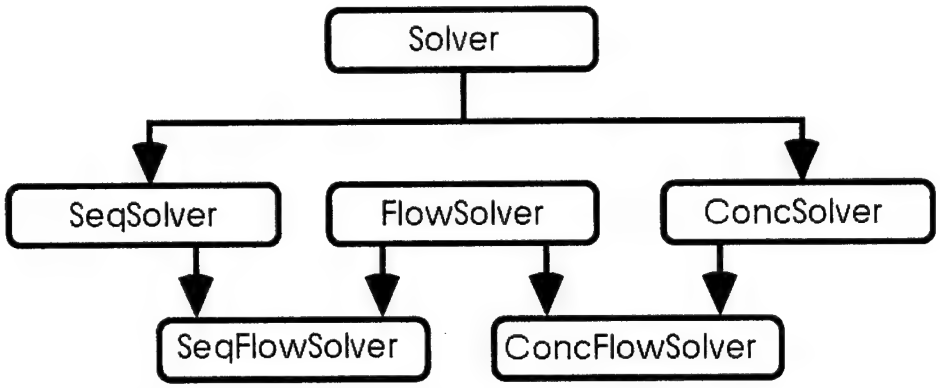

Figure 7: *Class hierarchy for* FlowSolver *and its specializations*

Instances of both SeqFlowSolver and ConcFlowSolver can be defined by reading records from an OFFSRF input file. Extractors are overloaded for the SeqFlowSolver and OFFSRF_ifstream classes, and for the ConcFlowSolver and OFFSRF_ifstream classes. Thus, if s is a SeqFlowSolver or a ConcFlowSolver and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> s;
```

will define s by reading the OFFSRF file associated with the stream in_stream. The extractor reads the following OFFSRF records.

1. A GRID record in the format described in Section 2.5.
2. A GLOBAL VARIABLES record which defines the organization of the independent variables. The format of this record is described below.
3. A REYNOLDS NUMBER record with format
   {REYNOLDS NUMBER: *number* }
   where *number* is the value of the Reynolds number.
4. Any number of records describing the sub-solvers. In the current version of TRANSOM the following OFFSRF records are recognized for defining the sub-solvers.

   SPC 2D SOLVER: Creates a solver which uses the pseudo-compressibility method to solve the Navier-Stokes equations on a structured grid. The format of this record is described in Reference 12.

   STRUCTURED STREAMFUNCTION SOLVER: Creates a solver which calculates a streamfunction for a two-dimensional flow field on a structured grid. Typically this solver is used in sequence with an SPC 2D SOLVER.

   FE 2D NS SOLVER: Creates a solver which solves the two-dimensional Navier-Stokes equations using the finite element method. The format of this record is described in Reference 13.

   FE STREAMFUNCTION SOLVER: Creates a solver which uses the finite element method to calculate a streamfunction for a two-dimensional

flow field. Typically this solver is used in sequence with an FE NS SOLVER.

FE LAPLACE SOLVER: Creates a solver which uses the finite element method to solve Laplace's equation.

SEQUENTIAL SOLVER: Creates a new instance of a SeqFlowSolver. This record should have a format similar to that of the TRANSOM input file except that it may not include a GRID record. If independent variables have been inherited from the parent solver, then it also must not contain a GLOBAL VARIABLES record.

CONCURRENT SOLVER: Creates a new instance of a ConcFlowSolver. The format of this record is exactly the same as for the SEQUENTIAL SOLVER record.

The format of the GLOBAL VARIABLES record is as follows:

{GLOBAL VARIABLES
name-1 *length-1*
name-2 *length-2*
⋮ ⋮
name-n *length-n*
}GLOBAL VARIABLES

where *name-1* to *name-n* are the names of the variables and *length-1* to *length-n* are the number of doubles required per node for the corresponding variable. If the variable names contain spaces, they must be enclosed in double quotes. For example, the following record defines three global variables: Pressure, two-dimensional Velocity, and Turbulent Viscosity.

```
{GLOBAL VARIABLES
  Pressure 1
  Velocity 2
  "Turbulent Viscosity" 1
}GLOBAL VARIABLES
```

The variables can also be initialized from within the GLOBAL VARIABLES record; simply include a record whose name is the name of the variable, followed by the values of that variable at all the nodes (note that this is the format required by the IOVar extractor as described in Section 3.2). Thus, the following record defines Pressure and two-dimensional Velocity and initializes the values of Pressure.

{GLOBAL VARIABLES
Pressure 1
Velocity 2
{Pressure
p_1 p_2 ⋯ p_n
}Pressure
}GLOBAL VARIABLES

To implement the extractors, a mechanism must be provided for defining the sequence of sub-solvers by reading OFFSRF records. Moreover, it should be possible for the sub-solvers themselves to be instances of the SeqFlowSolver

and ConcFlowSolver. To avoid duplication of code between the two solvers, the task of reading the input records is relegated to the base class FlowSolver. The FlowSolver virtual member function insert(ConnectedSolver*) is used to insert the sub-solver into the list of sub-solvers; for the class SeqFlowSolver this function is simply redefined as SeqSolver::insert(ConnectedSolver*), while for the ConcFlowSolver it is redefined as ConcSolver::insert(ConnectedSolver*).

Whenever a sub-solver is created by a FlowSolver, the grid (or perhaps an appropriate block of the grid) and the independent variables of the FlowSolver are passed to the sub-solver via its constructor. All the sub-solvers then share the grid and the variables of their parent. If a sub-solver requires an independent variable which is not defined by the parent (e.g. if the solver requires turbulent viscosity but the parent defines only pressure and velocity), then a local copy of that variable will normally be generated.

4.6 Residual Solvers

Residual solvers are an important abstraction of solvers used to solve systems of equations. Consider the system of equations

$$f(x) = 0$$

where x is a state vector and f is a vector of functions. In a residual solver, one evaluates $r = f(x)$ for the current approximation to the state vector x, then determines a correction to x which will reduce the magnitude of the vector r. The elements of r are called the residuals of the system of equations. The solution algorithm can be written as follows.

$$\begin{aligned} S(x) = \{\ & x \leftarrow x_0 \\ & \text{repeat} \\ & \quad \{\ r \leftarrow f(x) \\ & \qquad x \leftarrow x + g(x,r) \\ & \quad \} \\ & \text{while not } C(r) \\ \} & \end{aligned} \tag{4}$$

where x_0 is the initial state vector, $g(x,r)$ is the correction to x, and $C(r)$ is a stop test which returns true when the residuals are sufficiently small. To conform to the solution algorithm (1) in Section 4, the post-tested loop must be converted to a pre-tested loop. This is done simply by requiring that the stop test always return false the first time it is called after initialization. Algorithm (1) may then be used with the sweep function defined by

$$\begin{aligned} F(x) = \{\ & r \leftarrow f(x) \\ & x \leftarrow x + g(x,r) \\ \} & \end{aligned} \tag{5}$$

In TRANSOM, residual solvers are implemented using the three classes

ResidualSolver, ResidualStopTest, and ResidualSweep. In these classes the residuals are represented as a Vec: an array of floating point values for which many arithmetic operators have been defined. The full definition of the Vec class may be found in the source files Mtx.h and Mtx.c. The state vector is represented as an IOVar.

4.6.1 The ResidualStopTest Class

The ResidualStopTest class represents a stop test suitable for use in a residual solver. The stop test always returns false the first time it is called after initialization. Otherwise it returns true if one of two conditions is satisfied.

1. The largest absolute value of all the residuals is less than the small predefined value ε_{acc}. This conditions stops the solver when the solution has converged.
2. The largest absolute value of all the residuals is greater than the large predefined value ε_{div}. This condition stops the solver when the solution has diverged.

The ResidualStopTest constructors and member functions are:

ResidualStopTest(Vec *r, double a, double maxa);
: Makes a ResidualStopTest with residuals obtained from r, with ε_{acc} set to a, and with ε_{div} set to maxa.

virtual void initialize();
: Initializes the stop test. The next call to stop() will return false.

virtual int stop();
: Executes the stop test.

void set_max_residual(double maxa);
: Sets the value of ε_{div} to maxa.

void set_acc(double a);
: Sets the value of ε_{acc} to a.

The ResidualStopTest class is a specialization of the ReportStopTest class. The member functions set_report(), report(), and set_report_string() are inherited from ReportStopTest. The report from the ResidualStopTest prints the maximum and the root mean square residual. The default string prepended to the report is "Accuracy = ", but it can be changed using set_report_string(). Thus, if rst is a ResidualStopTest, then the report is similar to the following.

```
Accuracy =      -0.123456      0.0256543
```

The first number is the residual with the largest absolute value and the second is the root mean square residual.

4.6.2 The ResidualSweep Class

The ResidualSweep class is used to implement the sweep algorithm (5). Its data includes a Vec to represent the residuals, and an IOVar, data, to represent

the state vector. It has the following constructors and member functions.

ResidualSweep(IOVar *d, unsigned nr = 0);
Makes a ResidualSweep with state vector d. The length of the residual vector is given by nr. If nr is zero, then the length of the residual vector is set to the number of doubles in d.

virtual void calc_residual() = 0;
Calculates the residuals. This is a pure virtual function which must be defined by the specializations of the ResidualSweep class.

virtual MultiNumVar& correct() = 0;
Returns the correction to the state vector $g(x,r)$. This is a pure virtual function which must be defined by the specializations of the ResidualSweep class.

virtual void sweep();
Executes the sweep algorithm (5) by calling calc_residual() and then incrementing the state vector by the returned value of correct().

An extractor is overloaded for the ResidualSweep and OFFSRF_ifstream classes. It reads in values of the state vector, data. Thus, if rsw is a ResidualSweep and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> rsw;
```

will define data by reading the OFFSRF file associated with the stream in_stream. This code is identical in function to

```
in_stream >> rsw->data;
```

The format for the records to define an IOVar is described in Section 3.4.

4.6.3 The ResidualSolver Class

Residual solvers are represented by the ResidualSolver class. The sweep of a ResidualSolver is a specialization of a ResidualSweep. Its stop test is an OrStopTest which combines an NTimesStopTest and a ResidualStopTest. Thus, the solver will stop either when the residuals are sufficiently small (or too large) or when a fixed number of sweeps have been performed. The ResidualSolver ensures that its sweep and the ResidualStopTest share the same residuals. The constructor and member functions of the ResidualSolver class are:

ResidualSolver(ResidualSweep *s, unsigned mxniter = 1000, double acc = 1.0e-04, double maxacc = 1.0e+04);
Makes a ResidualSolver having sweep s. A maximum of mxniter sweeps will be performed. The solver stops executing sweeps when the maximum residual is smaller than acc or if it is greater than maxacc.

void set_max_residual(double maxacc)
Sets the value of ε_{div} for the ResidualStopTest to maxacc.

void set_acc(double acc);
Sets the value of ε_{acc} for the ResidualStopTest to acc.

void set_max_number_sweeps(unsigned mxniter);
Sets the maximum number of sweeps to mxniter.

The values of ε_{acc} and ε_{div} can be defined by reading records from an OFFSRF input file. An extractor is overloaded for the ResidualSolver and OFFSRF_ifstream classes. Thus, if rslv is a ResidualStopTest and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> rslv;
```

will define st by reading the OFFSRF file associated with the stream in_stream. The records should have the following format (all records are optional).

{RESIDUAL ACCURACY: *eps_acc* }
{MAXIMUM RESIDUAL: *eps_div* }
{MAXIMUM NUMBER ITERATIONS: *num* }

where *eps_acc* is the value of ε_{max}, *eps_div* is the value of ε_{div} and *num* is the maximum number of iterations allowed. Any other records will be passed to the sweep for interpretation.

5 The Main Program

The TRANSOM main program is very simple. It consists of four phases.

1. First a ConcFlowSolver, ms, is created.
2. Then an OFFSRF file is opened (its name is supplied as a command line argument when TRANSOM is executed from the shell) and the ConcFlowSolver extractor is used to define ms according to the contents of the file. As described in Section 4.5, this will define the grid, any global variables to be shared among sub-solvers, and all the sub-solvers to be executed.
3. Next ms.solve() is called to generate a solution to the problem.
4. Finally an OFFSRF output file is opened (its name is supplied as a command line argument when TRANSOM is executed from the shell) and the solution is written to it using the ConcFlowSolver inserter.

6 Concluding Remarks

TRANSOM is a complex program which is still in the early stage of development. There is little doubt that some of the structure defined in this memorandum will be changed as that development continues. Nevertheless, in the two flow solvers which have been implemented so far, the current class hierarchies have proved their worth.

At present TRANSOM works well when either the pseudo-compressibility solver or the finite element solver is used on a single block. Experimentation with a single solution method on multiple blocks has also shown that acceptable levels of convergence can be achieved. Some flows have also been solved in which both the pseudo-compressibility method and

the finite element method have been used on neighbouring blocks[17], although convergence is at best slow and at worst not attained. Work in this area will continue over the coming year; it will likely lead to significant changes in the ConnectedSolver and SweepConnection classes currently used to implement the transfer of information between solvers on neighbouring blocks.

Further development will also continue on each of the solvers implemented so far. Details of their shortcomings and the modifications proposed are described in References 12 and 13.

References

1. D. Hally, "User's Guide for HLLFLO Version 2.0", DREA Technical Memorandum 93/309, 1993.
2. D. Hally, "Implementation of a Free Surface in Calculations of the Flow into the Propeller Plane of a Ship", in *Proceedings of CADMO92: Third International Conference on Computer Aided Design Manufacture and Operation in the Marine and Offshore Industries*, Madrid, Spain, October 1992.
3. D. J. Hall and D. W. Zingg and C. R. Ethier, "A Laminar Incompressible Navier-Stokes Flow Solver", DREA Contractor Report CR/93/462, 1993.
4. D. J. Hall and D. W. Zingg and C. R. Ethier, "A Two-Dimensional Incompressible Navier-Stokes Turbulent Flow Solver", DREA Contractor Report, CR/94/435, 1994.
5. D. Hally, "Revisions to NSI2D", DREA Technical Memorandum 94/303, 1994.
6. S. E. Rogers and D. Kwak, "An Upwind Differencing Scheme for the Incompressible Navier-Stokes Equations," NASA TN 101051, Nov. 1988.
7. B. S. Baldwin and H. Lomax, "Thin Layer Approximation and Algebraic Model for Separated Turbulent Flows," AIAA Paper 78-257, 1978.
8. H. C. Chen and V. C. Patel, "Near-Wall Turbulence Models for Complex Flows Including Separation," AIAA Journal, Vol. 26, No. 6, pp. 641-648, June 1988.
9. B. S. Baldwin and T. J. Barth, "A One-Equation Turbulence Transport Model for High Reynolds Number Wall-Bounded Flows," NASA TM 102847, August 1990.
10. G. Dhatt and G. Touzot, *Une Présentation de la Méthode des Eléments Finis*, S. A. Maloine, ed., Paris, 1981.
11. F. Pineau, "A Finite Element Based Reynolds Averaged Navier-Stokes Solver for Modelling Three-Dimensional Turbulent Flows," DREA Report 93/106, 1993.
12. D. Hally, "Implementation of a Pseudo-Compressibility Navier-Stokes Solver in TRANSOM", DREA Technical communication, in preparation.
13. D. Hally, "Implementation of a Finite Element Navier-Stokes Solver in TRANSOM", DREA Technical communication, in preparation.
14. D. Hally, "User's Guide for TRANSOM Version 1.0: A Two-Dimensional Multi-Method Reynolds-Averaged Navier-Stokes Solver", DREA Technical Memorandum, in preparation.
15. D. Hally, "OFFSRF: A System for Representing Ship Offset Data", DREA Technical Memorandum 89/305, 1989.

16. D. Hally, "C++ Classes for Reading and Writing files in OFFSRF Format", DREA Technical Memorandum 94/302, 1994.
17. D. Hally, "Experiments with mixed-method flow solutions using TRANSOM", DREA Note, in preparation.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence Research Establishment Atlantic
P.O. Box 1012, Dartmouth, N.S. B2Y 3Z7

2. SECURITY CLASSIFICATION (Overall security of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title.)

TRANSOM: A Multi-Method Reynolds-Averaged Navier-Stokes Solver: Overall Design

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

HALLY, David

5. DATE OF PUBLICATION (Month and year of publication of document.)

June 1996

6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)

46

6b. NO. OF REFS. (Total cited in document.)

17

6. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the reseach and development. include the address.)

Defence Research Establishment Atlantic
P.O. Box 1012, Dartmouth, N.S. B2Y 3Z7

9a. PROJECT OR GRANT NUMBER (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

1.g.a.

9b. CONTRACT NUMBER (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number mrst be unique to this document.)

DREA Technical Memorandum 97/231

10b. OTHER DOCUMENT NUMBERS (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; futher distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic annoucement of this document. This will normally correspond to the Document Availability (11). However, where futher distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

TRANSOM is a multi-block, multi-method Reynolds-Averaged Navier Stokes solver being developed at DREA to address problems associated with the flow around ships and submarines. It is multi-block because the flow is divided into several distinct regions. It is multi-method because a different solution method may be used on each of the flow regions. At present two different methods of solution can be chosen; a finite-volume solver based on the pseudo-compressibility method; and a finite element solver which uses the penalty function method to determine the pressure.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the overall design of TRANSOM with emphasis on the class hierarchies used to represent different types of blocks and flow solvers. The algorithms used to implement the pseudo-compressibility and finite element methods are not discussed here; two companion reports describe these sub-solvers in detail.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Fluid flow
Turbulence
Reynolds-Averaged Navier-Stokes Equations
Hydrodynamics
Computer programs
Object Oriented Design
TRANSOM